Kapitel 4: NumPy-Grundlagen

McKinney, W. (2017). *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*. 2. Auflage. Sebastopol, CA [u. a.]: O'Reilly.

Überarbeitet durch armin.baenziger@zhaw.ch. Letzte Anpassung: 13.01.2020

- NumPy, kurz für Numerical Python, ist ein zentrales Packet für numerische Berechnungen in Python.
- In diesem Kapitel werden **Arrays** und das Konzept der **"Vektorisierung"** eingeführt.
- Da Pandas auf NumPy aufbaut, sind Grundlagenkenntnisse in NumPy hilfreich.

In [1]:
```
%autosave 0
```

```
Autosave disabled
```

NumPys Ndarray: Ein mehrdimensionales Array-Objekt

Zuerst laden wir die NumPy-Bibliothek mit der üblichen Abkürzung np:

In [2]:
```
import numpy as np
```

- Eine wichtige Datenstruktur in NumPy ist der N-dimensionale Array, oder **`ndarray`**, welcher ein schneller, flexibler Container für grosse Datensets in Python ist.
- Es folgt ein Beispiel von einem Ndarray:

In [3]:
```
arr1 = np.array([[6, 3, 5], [3, 2, 4]])
arr1
```

Out[3]:
```
array([[6, 3, 5],
       [3, 2, 4]])
```

`arr1` ist *zwei*dimensional. In der Datenanalyse sind ein- und zweidimensionale Arrays üblich.

In [4]:
```
arr1.ndim     # Dimension des Array
```

Out[4]:
```
2
```

In [5]:
```
arr1.shape    # arr1 hat zwei Zeilen und drei Spalten
```

Out[5]:
```
(2, 3)
```

- **Mit Arrays können mathematische Operationen auf ganze Datenblöcke angewendet werden.**
- **Die Syntax ist dabei ähnlich wie bei den entsprechenden Operationen zwischen Skalaren.**
- Beispiel:

In [6]:
```
arr1 + 3
```

Out[6]:
```
array([[9, 6, 8],
       [6, 5, 7]])
```

```
In [7]: arr1 * 10
```

```
Out[7]: array([[60, 30, 50],
               [30, 20, 40]])
```

Ein `ndarray` ist ein *multidimensionaler* Container für *homogene* Daten (Daten vom gleichen Typ).

```
In [8]: arr1.dtype     # Datentyp im Array
```

```
Out[8]: dtype('int32')
```

Ndarrays erstellen

Listen können mit der Funktion `array` einfach in Arrays umgewandelt werden.

```
In [9]: liste1 = [6, 7.5, 8, 0, 1]
        arr1 = np.array(liste1)
        arr1
```

```
Out[9]: array([6. , 7.5, 8. , 0. , 1. ])
```

```
In [10]: arr1.ndim   # eindimensionaler Array (Vektor)
```

```
Out[10]: 1
```

```
In [11]: liste2 = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
         arr2 = np.array(liste2)
         arr2
```

```
Out[11]: array([[ 1,  2,  3,  4],
                [ 5,  6,  7,  8],
                [ 9, 10, 11, 12]])
```

```
In [12]: arr2.ndim    # Zweidimensionaler Array (Matrix)
```

```
Out[12]: 2
```

```
In [13]: arr2.shape   # Atribut shape: Der Array hat 3 Zeilen und 4 Spalten.
```

```
Out[13]: (3, 4)
```

Mit der Methode `reshape` kann ein Array in eine neue Form gebracht werden.

```
In [14]: arr2.reshape((2,6))  # in 2x6-Array umwandeln
```

```
Out[14]: array([[ 1,  2,  3,  4,  5,  6],
                [ 7,  8,  9, 10, 11, 12]])
```

Spezielle Arrays erstellen:

Die np-Funktion `arrange` entspricht der `range` -Funktion in Core-Python.

```
In [15]: np.arange(10)
```

```
Out[15]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [16]:
```
np.arange(-3, 12, 2)    # von -3 bis unter 12 mit Schritt von 2
```

Out[16]:
```
array([-3, -1,  1,  3,  5,  7,  9, 11])
```

In [17]:
```
np.arange(101, 113).reshape(2,6)
```

Out[17]:
```
array([[101, 102, 103, 104, 105, 106],
       [107, 108, 109, 110, 111, 112]])
```

Arrays mit lauter Nullen erstellen:

In [18]:
```
arr1_zeros = np.zeros(10)
arr1_zeros
```

Out[18]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

In [19]:
```
arr2_zeros = np.zeros((2,5))
arr2_zeros
```

Out[19]:
```
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

Arrays mit lauter Einsen erstellen:

In [20]:
```
np.ones((3, 6))
```

Out[20]:
```
array([[1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1.]])
```

Anlegung eines Arrays ohne speziellen Inhalt (Container für später hinzuzufügenden Inhalt):

In [21]:
```
container = np.empty((2, 5))
container
# Achtung: empty liefert nicht unbedingt nur 0, sondern
# uninitialisierten "Müll".
```

Out[21]:
```
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

Kontrollfragen:

In [22]:
```
# Gegeben:
array1 = np.array([3, 4, 1, 0, 2, 2])
array1
```

Out[22]:
```
array([3, 4, 1, 0, 2, 2])
```

In [23]:
```
# Frage 1: Was ist der Output?
array1.ndim
```

Out[23]:
```
1
```

In [24]:
```
# Frage 2: Erzeugen Sie aus dem Array "array1" den Array
# "array2" mit drei Zeilen und zwei Spalten. Verwenden Sie
# hierzu die Methode reshape().
array2 = array1.reshape(3,2)
array2
```

Out[24]:
```
array([[3, 4],
       [1, 0],
       [2, 2]])
```

In [25]:
```
# Frage 3: Was ist der Output?
array2.shape
```

Out[25]:
```
(3, 2)
```

Mehr zur Arithmetik mit NumPy-Arrays

- Arrays sind wichtig, weil man mit ihnen viele Batch-Operationen ohne `for` -Loops umsetzen kann. Man spricht von *Vektorisierung (vectorization)*.
- Arithmetische Operationen zwischen Arrays mit gleichen Dimensionen (bzw. Shape ist identisch) führen dazu, dass die Operationen **elementweise** durchgeführt werden.

In [26]:
```
# Gegeben:
arr1 = np.array([[1, 2, 3], [4, 5, 6]])
arr1
```

Out[26]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [27]:
```
# Gegeben:
arr2 = np.array([[0, 2, 1], [1, 0, 3]])
arr2
```

Out[27]:
```
array([[0, 2, 1],
       [1, 0, 3]])
```

In [28]:
```
arr1 + arr2   # elementweise Addition
```

Out[28]:
```
array([[1, 4, 4],
       [5, 5, 9]])
```

In [29]:
```
arr1 ** arr2  # elementweise Potenz
```

Out[29]:
```
array([[  1,   4,   3],
       [  4,   1, 216]], dtype=int32)
```

Operationen mit einem Skalar ("einzelner Wert") führen dazu, dass der Skalar auf jedes Element des Arrays übertragen wird. Man spricht von *"Broadcasting"*.

In [30]:
```
arr1 * 1.1
```

Out[30]:
```
array([[1.1, 2.2, 3.3],
       [4.4, 5.5, 6.6]])
```

In [31]:
```
arr1 ** 0.5  # Quadratwurzel aller Elemente
```

Out[31]:
```
array([[1.        , 1.41421356, 1.73205081],
       [2.        , 2.23606798, 2.44948974]])
```

Elementweise Multiplikation:

In [32]:
```
arr1 * arr2
```

Out[32]:
```
array([[ 0,  4,  3],
       [ 4,  0, 18]])
```

Hinweis für diejenigen, die sich mit linearer Algebra auskennen: Die eigentliche Matrixmultiplikation wird mit `np.dot()` oder `@` umgesetzt.

Kontrollfragen:

In [33]:
```
# Gegeben:
arr1
```

Out[33]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [34]:
```
# Gegeben:
arr2
```

Out[34]:
```
array([[0, 2, 1],
       [1, 0, 3]])
```

In [35]:
```
# Frage 1: Was ist der Output?
arr1.shape
```

Out[35]:
```
(2, 3)
```

In [36]:
```
# Frage 2: Was ist der Output?
arr1 - 3
```

Out[36]:
```
array([[-2, -1,  0],
       [ 1,  2,  3]])
```

In [37]:
```
# Frage 3: Was ist der Output?
arr2 * 3
```

Out[37]:
```
array([[0, 6, 3],
       [3, 0, 9]])
```

In [38]:
```
# Frage 4: Was ist der Output?
arr1 - arr2
```

Out[38]:
```
array([[1, 0, 2],
       [3, 5, 3]])
```

Grundlegende Indexierung und Slicing

Zur Erinnerung: Python beginnt die Indexierung mit 0 (1. Wert in der Sequenz).

In [39]:
```
arr = np.arange(10, 20)
arr
```

Out[39]:
```
array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19])
```

In [40]:
```
arr[5]  # Indexierung wie gewohnt
```

Out[40]:
```
15
```

In [41]:
```
arr[-1] # Letzter Wert der Sequenz
```

Out[41]:
```
19
```

In [42]:
```
arr[:3]  # Slicing wie gewohnt
```

Out[42]:
```
array([10, 11, 12])
```

In [43]:
```
arr[5:8]
```

Out[43]:
```
array([15, 16, 17])
```

In [44]:
```
arr[5:8] = 0
arr
```

Out[44]:
```
array([10, 11, 12, 13, 14,  0,  0,  0, 18, 19])
```

Ein wichtiger Unterschied zwischen (built-in) Python-Listen und NumPy-Arrays liegt darin, dass Array-Sclices **Views** (und keine Kopie) auf den Original-Array darstellen.

In [45]:
```
arr_slice = arr[5:8]
arr_slice
```

Out[45]:
```
array([0, 0, 0])
```

In [46]:
```
arr_slice[1] = 999
arr               # Auch das Original wurde verändert!
```

Out[46]:
```
array([ 10,  11,  12,  13,  14,   0, 999,   0,  18,  19])
```

Will man explizit eine Kopie (und keinen View), muss man die `copy` -Methode verwenden.

In [47]:
```
arr_slice = arr[5:8].copy()
arr_slice[1] = 1000
arr_slice
```

Out[47]:
```
array([   0, 1000,    0])
```

In [48]:
```
arr         # Das Original wurde nun nicht verändert!
```

Out[48]:
```
array([ 10,  11,  12,  13,  14,   0, 999,   0,  18,  19])
```

Indexierung und Slicing bei Arrays mit höheren Dimensionen

Zwei Dimensionen

In [49]:
```
arr2d = np.arange(9).reshape(3,3)
arr2d
```

Out[49]:
```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

In [50]:
```
arr2d[2]        # 3. Zeile (Index 2)
```

Out[50]:
```
array([6, 7, 8])
```

In [51]:
```
arr2d[0][1]     # Rekursive Indexierung: 1. Zeile, 2. Spalte
```

Out[51]:
```
1
```

In [52]:
```
arr2d[0, 1]     # Zweite Möglichkeit: 1. Zeile, 2. Spalte
```

Out[52]:
```
1
```

Bei zweidimensionalen Arrays (Matrizen) gilt der Merksatz: **Zeilen zuerst, Spalten später**

In [53]:
```
arr2d[2,:]      # Alternative zu arr2d[2]
```

Out[53]:
```
array([6, 7, 8])
```

In [54]:
```
arr2d[:,1]      # 2. Spalte auswählen
```

Out[54]:
```
array([1, 4, 7])
```

In [55]:
```
arr2d[:2, 1:] # Die ersten zwei Zeilen und ab zweiter Spalte
```

Out[55]:
```
array([[1, 2],
       [4, 5]])
```

In [56]:
```
arr2d[1, :2]  # Zweite Zeile und die ersten zwei Spalten
```

Out[56]:
```
array([3, 4])
```

In [57]:
```
arr2d[:2, 2]
```

Out[57]:
```
array([2, 5])
```

In [58]:
```
arr2d[:2, 1:] = 0
arr2d
```

Out[58]:
```
array([[0, 0, 0],
       [3, 0, 0],
       [6, 7, 8]])
```

Kontrollfragen:

In [59]:
```
# Gegeben:
array3 = np.arange(11,20).reshape(3,3)
array3
```

Out[59]:
```
array([[11, 12, 13],
       [14, 15, 16],
       [17, 18, 19]])
```

In [60]:
```
# Frage 1: Was ist der Output?
array3[1,2]
```

Out[60]:
```
16
```

In [61]:
```
# Frage 2: Was ist der Output?
array3[:2]
```

Out[61]:
```
array([[11, 12, 13],
       [14, 15, 16]])
```

In [62]:
```
# Frage 3: Was ist der Output?
array3[:2, 1:]
```

Out[62]:
```
array([[12, 13],
       [15, 16]])
```

Advanced Indexing

Darunter fallen boolsche Indexierung und Ganzzahlindexierung.

Boolsche Indixierung

Grundsätzlicher Mechanismus:

In [63]:
```
x = np.array([21, -7, 19, -2, 0])
x
```

Out[63]:
```
array([21, -7, 19, -2,  0])
```

In [64]:
```
bool_arr = np.array([True, True, False, True, False])
bool_arr
```

Out[64]:
```
array([ True,  True, False,  True, False])
```

In [65]:
```
x[bool_arr]
# Diejenigen Werte von x werden ausgewählt, an deren Stelle True steht.
```

Out[65]:
```
array([21, -7, -2])
```

Üblicherweise generieren wir den boolschen Array über eine Bedingung:

In [66]:
```
bool_arr = (x > 0)
bool_arr
```

Out[66]:
```
array([ True, False,  True, False, False])
```

In [67]:
```
x[bool_arr] # Alle x-Werte grösser 0
```

Out[67]:
```
array([21, 19])
```

In [68]:
```
x[x > 0]    # Man kann die Bedingung auch direkt übergeben.
```

Out[68]:
```
array([21, 19])
```

Weitere Beispiele:

In [69]:
```
zivilstand = np.array(['ledig', 'geschieden', 'verheiratet', 'ledig'])
```

In [70]:
```
alter = np.array([27, 55, 35, 18])
```

In [71]:
```
ist_ledig = (zivilstand == 'ledig')  # boolscher Array
```

In [72]:
```
alter[ist_ledig]     # Alter von ledigen Personen
```

Out[72]:
```
array([27, 18])
```

In [73]:
```
# Es ginge auch direkt:
alter[zivilstand == 'ledig']
```

Out[73]:
```
array([27, 18])
```

In [74]:
```
alter[ist_ledig][0]  # Alter der ersten ledigen Person
```

Out[74]:
```
27
```

In [75]:
```
alter[zivilstand != 'ledig']     # Alter von nicht ledigen Personen
```

Out[75]:
```
array([55, 35])
```

In [76]:
```
alter[~(zivilstand == 'ledig')] # geht auch so
```

Out[76]:
```
array([55, 35])
```

Der Operator " ~ " ist sehr hilfreich, um einen (bereits erstellten) boolschen Array zu invertieren.

In [77]:
```
alter[~ist_ledig]    # Das Alter aller Personen, die NICHT ledig sind.
```

Out[77]:
```
array([55, 35])
```

Bedingungen können auch kombiniert werden:

In [78]:
```
auswahl = (zivilstand == 'ledig') | (zivilstand == 'geschieden')
auswahl
```

Out[78]:
```
array([ True,  True, False,  True])
```

In [79]:
```
alter[auswahl]
```

Out[79]:
```
array([27, 55, 18])
```

- Die Schlüsselwörter `and` und `or` funktionieren nicht mit boolschen Arrays. Man benützt `&` (und) und `|` (oder).
- Boolsche Indexierung funktioniert auch für Arrays höherer Dimension:

In [80]:
```
data = np.array([[9, -2], [7, 0], [-4, 1]])
data
```

Out[80]:
```
array([[ 9, -2],
       [ 7,  0],
       [-4,  1]])
```

In [81]:
```
auswahl = [True, False, True]
```

In [82]:
```
data[auswahl]  # erste Zeile ja, zweite nein, dritte ja
```

Out[82]:
```
array([[ 9, -2],
       [-4,  1]])
```

In [83]:
```
data < 0     # welche Elemente sind negativ?
```

Out[83]:
```
array([[False,  True],
       [False, False],
       [ True, False]])
```

In [84]:
```
data[data < 0]     # negative Elemente auswählen
```

Out[84]:
```
array([-2, -4])
```

In [85]:
```
data[data < 0] = 0 # Alle negativen Werte mit 0 ersetzen.
data
```

Out[85]:
```
array([[9, 0],
       [7, 0],
       [0, 1]])
```

Kontrollfragen:

In [86]:
```
# Gegeben:
print('array1:', array1)
array2 = np.array(list('abcdef'))
print('array2:', array2)
```

```
array1: [3 4 1 0 2 2]
array2: ['a' 'b' 'c' 'd' 'e' 'f']
```

In [87]:
```
# Frage 1: Was ist der Output?
array2[[True, False, True, False, False, False]]
```

Out[87]:
```
array(['a', 'c'], dtype='<U1')
```

In [88]:
```
# Frage 2: Was ist der Output?
array2[array1 == 2]
```

Out[88]:
```
array(['e', 'f'], dtype='<U1')
```

In [89]:
```
# Frage 3: Was ist der Output?
array2[array1 > 0]
```

Out[89]:
```
array(['a', 'b', 'c', 'e', 'f'], dtype='<U1')
```

In [90]:
```
# Frage 4: Was ist der Output?
array1[(array2 == 'b') | (array2 == 'f')]
```

Out[90]:
```
array([4, 2])
```

Ganzzahlindexierung (Integer Indexing, auch Fancy Indexing)

Fancy Indexing ist ein NumPy-Begriff für das Indexieren mit Ganzzahlen-Arrays.

In [91]:
```
# Beispieldaten:
array2
```

Out[91]:
```
array(['a', 'b', 'c', 'd', 'e', 'f'], dtype='<U1')
```

In [92]:
```
array2[0]   # erster Wert (Repetition)
```

Out[92]:
```
'a'
```

In [93]:
```
# Auswahl von mehreren Werten mit gewünschter Reihenfolge:
array2[[0, 2, 1, 0]]
```

Out[93]:
```
array(['a', 'c', 'b', 'a'], dtype='<U1')
```

Beachten Sie, dass eine Liste (zwischen den eckigen Klammern) übergeben wird und dass auch Elemente wiederholt ausgewählt werden können.

In [94]:
```
# Weitere Beispieldaten:
arr = np.arange(5*3).reshape((5, 3))
arr
```

Out[94]:
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

In [95]:
```
arr[0]   # Repetition: Auswahl der ersten Zeile
```

Out[95]:
```
array([0, 1, 2])
```

In [96]:
```
arr[0,:]   # es geht auch so (erste Zeile und alle Spalten)
```

Out[96]:
```
array([0, 1, 2])
```

In [97]:
```
# Auswahl von mehreren Zeilen mit gewünschter Reihenfolge:
arr[[3, 0, 3]]  # Die Zeile 3 (vierte) wird zweimal ausgewählt!
```

Out[97]:
```
array([[ 9, 10, 11],
       [ 0,  1,  2],
       [ 9, 10, 11]])
```

In [98]:
```
arr[-1]  # letzte Zeile
```

Out[98]:
```
array([12, 13, 14])
```

In [99]:
```
arr[:,[0,2]]     # alle Zeilen und Spalten 0 und 2 auswählen
```

Out[99]:
```
array([[ 0,  2],
       [ 3,  5],
       [ 6,  8],
       [ 9, 11],
       [12, 14]])
```

- *Hinweis:* "Advanced Indexing" erstellt *eine Kopie der Daten*, selbst wenn der zurückgegebene Array unverändert ist (während Slicing einen View auf die Originaldaten erstellt).

Kontrollfragen:

In [100]:
```
# Gegeben:
array2
```

Out[100]:
```
array(['a', 'b', 'c', 'd', 'e', 'f'], dtype='<U1')
```

In [101]:
```
# Frage 1: Was ist der Output?
array2[[0, 0, 1]]
```

Out[101]:
```
array(['a', 'a', 'b'], dtype='<U1')
```

In [102]:
```
# Frage 2: Was ist der Output?
array2[[-1, -3]]
```

Out[102]:
```
array(['f', 'd'], dtype='<U1')
```

Arrays transponieren

Wir beschränken uns hier auf Arrays mit zwei Dimensionen bzw. Matrizen.

In [103]:
```
arr = np.arange(4*2).reshape((4, 2))
arr
```

Out[103]:
```
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7]])
```

In [104]:
```
# Zeilen werden zu Spalten und umgekehrt:
arr.transpose()   # Transpose-Methode
```

Out[104]:
```
array([[0, 2, 4, 6],
       [1, 3, 5, 7]])
```

In [105]:
```
arr.T           # kürzer mit Transpose-Attribut
```

Out[105]:
```
array([[0, 2, 4, 6],
       [1, 3, 5, 7]])
```

Universal Functions

Eine "Ufunc" ist eine Funktion, welche ***elementweise*** Operationen auf *Ndarrays* ausführt.

In [106]:
```
arr = np.arange(5)
arr
```

Out[106]:
```
array([0, 1, 2, 3, 4])
```

In [107]:
```
np.sqrt(arr)            # Quadratwurzel von jedem Element
```

Out[107]:
```
array([0.        , 1.        , 1.41421356, 1.73205081, 2.        ])
```

In [108]:
```
np.sqrt(arr).round(3)     # auf 3 Nachkommastellen gerundet
```

Out[108]:
```
array([0.   , 1.   , 1.414, 1.732, 2.   ])
```

In [109]:
```
np.exp(arr).round(3)    # Exponentialfunktion
```

Out[109]:
```
array([ 1.   ,  2.718,  7.389, 20.086, 54.598])
```

Binary Ufuncs (Ufuncs auf *zwei* Arrays)

In [110]:
```
x = np.array([1, 2, 5])
y = np.array([3, -7, 0])
np.maximum(x, y)
# Nimmt jeweils das Maximum von x und y pro Indexposistion
```

Out[110]:
```
array([3, 2, 5])
```

Array-basiertes Programmieren

- Viele Datenverarbeitungsaufgaben können statt mit Loops mit kurzen Array-Ausdrücken umgesetzt werden.
- Man spricht von *Vektorisierung*.
- Die Vektorisierung führt dazu, dass Operationen viel schneller ausgeführt werden als mit Loops.
- **Einfaches Beispiel**: Die Werte von zwei Listen sollen posisitionsbezogen zusammenaddiert werden und in einer dritten Liste abgespeichert werden.

In [111]:
```
# Gegeben:
list1 = [1, 5, -1, 0, 3]
list2 = [2, 3, 0, -2, 6]
```

Man könnte einen for-Loop oder eine List-Comprehension zur Lösung nutzen. Einfacher geht es mit NumPy-Arrays:

In [112]:
```
arr1 = np.array(list1)
arr2 = np.array(list2)

arr3 = arr1 + arr2
arr3   # oder list(arr3), falls eine Liste verlangt ist
```

Out[112]:
```
array([ 3,  8, -1, -2,  9])
```

Bedingungen als Array-Operationen formulieren

Angenommen wir haben zwei Werte-Arrays und einen boolschen Array:

In [113]:
```
cond = np.array([True, False, True, True, False])
print('arr1:', arr1)
print('arr2:', arr2)
print('cond:', cond)
```

```
arr1: [ 1  5 -1  0  3]
arr2: [ 2  3  0 -2  6]
cond: [ True False  True  True False]
```

Es soll nun ein Wert von `arr1` genommen werden, wenn der entsprechende Wert in `cond` `True` ist, ansonsten soll der Wert von `arr2` stammen.

Am einfachsten setzt man hierzu auf die Funktion `np.where`, welche der Excel-Funktion `WENN` gleicht.

In [114]:
```
result = np.where(cond, arr1, arr2)
result
```

Out[114]:
```
array([ 1,  3, -1,  0,  6])
```

Weitere Beispiele, wie man `where` einsetzen kann bei Datenanalysen:

In [115]:
```
np.where(arr1 > arr2, arr1, arr2)
# Nimmt jeweils den grösseren (oder gleich grossen) Wert der beiden Arrays.
```

Out[115]:
```
array([2, 5, 0, 0, 6])
```

Kontrollfrage:

In [116]:
```
# Frage: Was ist der Output?
np.where([True, False, True], [1, 2, 3], [10, 20, 30])
```

Out[116]:
```
array([ 1, 20,  3])
```

Mathematische und statistische Methoden

- Mit NumPy können (Quasi-) Zufallszahlen (aus verschiedenen Verteilungen) generiert werden.
- Wenn wir "Zufallszahlen" generieren, können wir einen sogenannten Seed setzten, damit wir reproduzierbare Ergebnisse erhalten (immer die gleichen Werte).

In [117]:
```
np.random.seed(327)
# Dadurch erhalten wir immer die gleichen "Zufallszahlen".
# In der Klammer kann auch eine andere natürliche Zahl
# zwischen 0 und 2**32 - 1 stehen!
```

In [118]:
```
arr = np.random.randn(2,3) # Standardnormaverteilte Zufallsvariablen
arr
```

Out[118]:
```
array([[ 1.71562172, -0.3980736 , -0.29765049],
       [-1.7918477 ,  0.97713881,  1.16761162]])
```

In [119]:
```
# Beispieldaten:
arr = np.random.randint(1, 7, 12)
# randint zieht ganze Zahlen, hier zwischen 1 bis (exklusiv) 7.
# Damit lässt sich z. B. (wie hier) ein Würfelsimulator erstellen.

arr    # 12 Würfelrealisationen
```

Out[119]:
```
array([2, 1, 3, 4, 1, 4, 2, 6, 6, 2, 6, 2])
```

Sogenannte *Aggregationen* (oder *Reduktionen*), wie `sum` (Summe), `mean` (arithm. Mittelwert) oder `var` (Varianz) kann man entweder als Methode oder np-Funktion aufrufen:

In [120]:
```
arr.sum()
```

Out[120]:
```
39
```

In [121]:
```
np.sum(arr)
```

Out[121]:
```
39
```

In [122]:
```
np.mean(arr)
```

Out[122]:
```
3.25
```

Die Aggregation kann über *alle* Werte des Arrays geschehen oder *entlang von Achsen* (Zeilen / Spalten).

In [123]:
```
# Beispieldaten:
np.random.seed(99)
arr = np.random.randint(0, 4, (3,4))
arr
```

Out[123]:
```
array([[1, 3, 1, 0],
       [1, 0, 2, 0],
       [1, 0, 1, 3]])
```

In [124]:
```
arr.sum()
# arithmetisches Mittel aller Werte im Array arr
```

Out[124]:
```
13
```

In [125]:
```
arr.sum(axis=0)
# Aggregation entlang von Zeilen (also Spaltenmittelwerte)
```

Out[125]:
```
array([3, 3, 4, 3])
```

In [126]:
```
arr.sum(axis=1)
# Aggregation entlang von Spalten (also Zeilenmittelwerte)
```

Out[126]:
```
array([5, 3, 5])
```

In [127]:
```
arr.mean(axis=1)  # Mittelwerte pro Zeile
```

Out[127]:
```
array([1.25, 0.75, 1.25])
```

Andere Methoden, wie `cumsum` und `cumpord` , aggregieren nicht sondern erzeugen einen neuen Array mit kumulierten Werten.

In [128]:
```
arr2 = np.array([0, 1, 3, 2, -1])
arr2.cumsum()
# Resultat: [0, 0+1, 0+1+3, 0+1+3+2, 0+1+3+2-1]
```

Out[128]:
```
array([0, 1, 4, 6, 5], dtype=int32)
```

Kontrollfragen:

In [129]:
```
# Gegeben:
arr
```

Out[129]:
```
array([[1, 3, 1, 0],
       [1, 0, 2, 0],
       [1, 0, 1, 3]])
```

In [130]:
```
# Frage 1: Wie erhalten wir die drei Spaltensummen von arr3.
arr.sum(axis=0)
```

Out[130]:
```
array([3, 3, 4, 3])
```

In [131]:
```
# Frage 2: Was erzeugt die folgende Anweisung?
np.array([2, 3, 1, 0]).cumsum()
```

Out[131]:
```
array([2, 5, 6, 6], dtype=int32)
```

Methoden für boolsche Arrays

In [132]:
```
# Vorbemerkung: True wird als 1 und False als 0 gerechnet:
True + False + True
```

Out[132]:
```
2
```

In [133]:
```
boolarray = np.array([True, False, True, False])
boolarray.sum()   # Anzahl True
```

Out[133]:
```
2
```

In [134]:
```
boolarray.any()    # Mindestens ein True?
```

Out[134]:
```
True
```

In [135]:
```
boolarray.all()      # Sind alle True?
```

Out[135]:
```
False
```

In [136]:
```
# Zur Erinnerung:
arr2
```

Out[136]:
```
array([ 0,  1,  3,  2, -1])
```

In [137]:
```
arr2 > 0            # Welche Elemente sind positiv?
```

Out[137]:
```
array([False,  True,  True,  True, False])
```

In [138]:
```
(arr2 > 0).sum()    # Anzahl positiver Werte
```

Out[138]:
```
3
```

In [139]:
```
print(arr1)
print(arr2)
```

```
[ 1  5 -1  0  3]
[ 0  1  3  2 -1]
```

In [140]:
```
# Ist an mind. einer Stelle arr1 grösser als arr2?
(arr1 > arr2).any()
```

Out[140]: True

Kontrollfragen:

In [141]:
```
# Gegeben:
arr
```

Out[141]:
```
array([[1, 3, 1, 0],
       [1, 0, 2, 0],
       [1, 0, 1, 3]])
```

In [142]:
```
# Frage 1: Was ist der Output?
(arr >= 3).sum()
```

Out[142]: 2

In [143]:
```
# Frage 2: Was ist der Output?
(arr == 0).mean()
```

Out[143]: 0.3333333333333333

In [144]:
```
# Frage: Was ist der Output?
(arr == 0).any()
```

Out[144]: True

Sortieren

In [145]:
```
# Ausgangslage:
arr2 = np.array([ 0,  1,  3,  2, -1])
```

In [146]:
```
sorted(arr2)   # arr2 bleibt erhalten.
```

Out[146]: [-1, 0, 1, 2, 3]

In [147]:
```
arr2
```

Out[147]: array([0, 1, 3, 2, -1])

In [148]:
```
np.sort(arr2)  # arr2 bleibt erhalten.
```

Out[148]: array([-1, 0, 1, 2, 3])

In [149]:
```
arr2
```

Out[149]: array([0, 1, 3, 2, -1])

In [150]:
```
arr2.sort()    # Sortierung ist permanent.
arr2
```

Out[150]:
```
array([-1,  0,  1,  2,  3])
```

Hinweis: Die Methode `sort` sortiert *inplace*. Die Funktion `np.sort` erstellt hingegen eine Kopie.

Mengenoperationen

Oft will man wissen, welche (oder auch wie viele) einzigartige Werte (unique values) vorhanden sind.

In [151]:
```
names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
np.unique(names)
```

Out[151]:
```
array(['Bob', 'Joe', 'Will'], dtype='<U4')
```

In [152]:
```
# Lösung mit Python-Grundfunktionen:
set(names)
```

Out[152]:
```
{'Bob', 'Joe', 'Will'}
```

In [153]:
```
# Anzahl unterschiedlicher Namen im Array names.
len(np.unique(names))
```

Out[153]:
```
3
```

Dateien einlesen und speichern

Da wir später Datensätze nur in Pandas einlesen und speichern, werden wir diesen Abschnitt im Lehrmittel nicht behandeln.

Lineare Algebra

Wird in diesem Kurs ebenfalls nicht thematisiert.

(Pseudo-) Zufallszahlengenerator

Zuvor haben wir bereits (ad hoc) Zufallszahlen mit NumPy erstellt. Es folgt ein Überblick und einige Ergänzungen

Für die Reproduzierbarkeit von Resultaten kann ein Seed gesetzt werden:

In [154]:
```
np.random.seed(987)
```

In [155]:
```
# Standardnormalverteilte Zufallsvariablen:
np.random.randn(3,2)
```

Out[155]:
```
array([[-1.68592706, -1.47111992],
       [-0.11180006,  1.03636979],
       [ 0.66103991,  0.88291241]])
```

In [156]:
```
# Alternative Funktion, die noch zusätzliche Argumente kennt:
np.random.normal(loc=5, scale=2, size=(3, 2))
# Normalverteilte Zufallsvariablen mit Mittelwert 5 und
# Standardabweichung 2 in einem 3x2-Array.
```

Out[156]:
```
array([[2.83334474, 3.30573661],
       [5.31655633, 5.19819463],
       [6.05362144, 3.93661013]])
```

In [157]:
```
# Uniformverteilung [0, 1):
np.random.rand(2, 3)
```

Out[157]:
```
array([[0.44472452, 0.76735829, 0.62039608],
       [0.67265825, 0.91189577, 0.8078284 ]])
```

In [158]:
```
# Diskrete Uniformverteilung:
np.random.randint(1, 7, (3, 4))  # z. B. Würfelaugen
```

Out[158]:
```
array([[4, 1, 4, 6],
       [2, 4, 1, 3],
       [6, 6, 3, 2]])
```

Im Lehrmittel werden weitere Funktionen vorgestellt, die wir aber im Kurs nicht benötigen.

Kontrollfragen:

In [159]:
```
# Aufgabe 1: Erstellen Sie den Vektor z mit 5 Realisationen aus einer
# Standardnormalverteilung.
z = np.random.randn(5)
z
```

Out[159]:
```
array([-1.53762265,  0.01595783, -1.48867534, -0.22020744,  0.90188405])
```

In [160]:
```
# Aufgabe 2: Erstellen Sie den Vektor z mit 1000 Realisationen aus einer
# Uniformverteilung (zwischen 0 und 1).
# Ermitteln Sie danach den Mittelwert von z.
z = np.random.rand(1000)
z.mean()
# Der Erwartungswert (Mittelwert, wenn n gegen unendlich geht) ist 0.5.
```

Out[160]:
```
0.4917437285165189
```

Fazit

- Während der Rest des Kurses sich darauf konzentriert, mit **Pandas** Data-Wrangling-Skills zu entwickeln, werden wir weiterhin mit einem ähnlichen Array-basierten Stil arbeiten.
- In Anhang A des Lehrmittels werden weiterführende NumPy-Kenntnisse vermittelt, auf die wir im Kurs nicht eingehen können.